UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
10/681,759	10/08/2003	Robert Anthony DeLine	MS304376.1	8309

27195   7590   07/10/2008
AMIN. TUROCY & CALVIN, LLP
24TH FLOOR, NATIONAL CITY CENTER
1900 EAST NINTH STREET
CLEVELAND, OH 44114

EXAMINER
YIGDALL, MICHAEL J

ART UNIT	PAPER NUMBER
2192	

NOTIFICATION DATE	DELIVERY MODE
07/10/2008	ELECTRONIC

Please find below and/or attached an Office communication concerning this application or proceeding.

The time period for reply, if any, is set in the attached communication.

Notice of the Office communication was sent electronically on above-indicated "Notification Date" to the following e-mail address(es):

docket1@thepatentattorneys.com
hholmes@thepatentattorneys.com
lpasterchek@thepatentattorneys.com

	Application No.	Applicant(s)
Office Action Summary	10/681,759	DELINE ET AL.
	Examiner	Art Unit	
	Michael J. Yigdall	2192	

-- The MAILING DATE of this communication appears on the cover sheet with the correspondence address --

Period for Reply

A SHORTENED STATUTORY PERIOD FOR REPLY IS SET TO EXPIRE _3_ MONTH(S) OR THIRTY (30) DAYS, WHICHEVER IS LONGER, FROM THE MAILING DATE OF THIS COMMUNICATION.
- Extensions of time may be available under the provisions of 37 CFR 1.136(a). In no event, however, may a reply be timely filed after SIX (6) MONTHS from the mailing date of this communication.
- If NO period for reply is specified above, the maximum statutory period will apply and will expire SIX (6) MONTHS from the mailing date of this communication.
- Failure to reply within the set or extended period for reply will, by statute, cause the application to become ABANDONED (35 U.S.C. § 133). Any reply received by the Office later than three months after the mailing date of this communication, even if timely filed, may reduce any earned patent term adjustment. See 37 CFR 1.704(b).

Status

1)☒ Responsive to communication(s) filed on _18 June 2008_.
2a)☐ This action is **FINAL**.   2b)☒ This action is non-final.
3)☐ Since this application is in condition for allowance except for formal matters, prosecution as to the merits is closed in accordance with the practice under *Ex parte Quayle*, 1935 C.D. 11, 453 O.G. 213.

Disposition of Claims

4)☒ Claim(s) _1-20 and 22-24_ is/are pending in the application.
    4a) Of the above claim(s) _____ is/are withdrawn from consideration.
5)☐ Claim(s) _____ is/are allowed.
6)☒ Claim(s) _1-20 and 22-24_ is/are rejected.
7)☐ Claim(s) _____ is/are objected to.
8)☐ Claim(s) _____ are subject to restriction and/or election requirement.

Application Papers

9)☐ The specification is objected to by the Examiner.
10)☐ The drawing(s) filed on _____ is/are: a)☐ accepted or b)☐ objected to by the Examiner.
    Applicant may not request that any objection to the drawing(s) be held in abeyance. See 37 CFR 1.85(a).
    Replacement drawing sheet(s) including the correction is required if the drawing(s) is objected to. See 37 CFR 1.121(d).
11)☐ The oath or declaration is objected to by the Examiner. Note the attached Office Action or form PTO-152.

Priority under 35 U.S.C. § 119

12)☐ Acknowledgment is made of a claim for foreign priority under 35 U.S.C. § 119(a)-(d) or (f).
    a)☐ All   b)☐ Some * c)☐ None of:
        1.☐ Certified copies of the priority documents have been received.
        2.☐ Certified copies of the priority documents have been received in Application No. _____.
        3.☐ Copies of the certified copies of the priority documents have been received in this National Stage application from the International Bureau (PCT Rule 17.2(a)).
    * See the attached detailed Office action for a list of the certified copies not received.

Attachment(s)
1) ☒ Notice of References Cited (PTO-892)
2) ☐ Notice of Draftsperson's Patent Drawing Review (PTO-948)
3) ☐ Information Disclosure Statement(s) (PTO/SB/08)
    Paper No(s)/Mail Date _____.
4) ☐ Interview Summary (PTO-413)
    Paper No(s)/Mail Date. _____ .
5) ☐ Notice of Informal Patent Application
6) ☐ Other: _____.

DETAILED ACTION

1.   A request for continued examination under 37 CFR 1.114, including the fee set forth in

37 CFR 1.17(e), was filed in this application after final rejection. Since this application is

eligible for continued examination under 37 CFR 1.114, and the fee set forth in 37 CFR 1.17(e)

has been timely paid, the finality of the previous Office action has been withdrawn pursuant to

37 CFR 1.114. Applicant's submission filed on June 19, 2008 has been entered. Claims 1-20

and 22-24 are pending.

Claim Rejections - 35 USC § 103

2.   The following is a quotation of 35 U.S.C. 103(a) which forms the basis for all

obviousness rejections set forth in this Office action:

> (a) A patent may not be obtained though the invention is not identically disclosed or described as set forth in
> section 102 of this title, if the differences between the subject matter sought to be patented and the prior art are
> such that the subject matter as a whole would have been obvious at the time the invention was made to a person
> having ordinary skill in the art to which said subject matter pertains. Patentability shall not be negatived by the
> manner in which the invention was made.

3.   Claims 1-6, 8-13, 15, 16 and 22-24 are rejected under 35 U.S.C. 103(a) as being

unpatentable over U.S. Patent No. 6,128,774 to Necula et al. (art of record, "Necula") in view of

U.S. Patent No. 6,353,925 to Stata et al. (now made of record, "Stata") and in view of Hallem et

al., "A System and Language for Building System-Specific, Static Analyses" (art of record,

"Hallem").

        With respect to claim 1 (currently amended), Necula discloses a computer implemented

executable code check system comprising:

an input component that receives an object file and a specification associated with the

object file (see FIGS. 3 and 4 – VCGen module 32 – and associated text, e.g., column 4, lines

37-42: "The VCGen module 32 performs two tasks. First, it checks simple safety properties of

the annotated executable 30. … Second, the VCGen module 32 watches for instructions whose

execution might violate the safety policy." The examiner notes that the object file is

synonymous with the executable file, and the specification associated with the object file is the

annotation embedded in the executable and/or rules defined by the safety policy.),

the specification comprising information associated with a plug-in condition for a method

(see FIG. 4 – configuration data 50 – and associated text, e.g., column 7, lines 19-32: "The

configuration data 50 also describes, by precondition-postcondition pairs, all of the functions that

the untrusted code 42 are permitted to invoke.").

Necula does not explicitly disclose that the plug-in condition parses contents of a query

string and makes the content available to a checker as part of a program's approximate execution

state.

However, in an analogous art, Stata teaches annotation processors that parse annotations

(see, for example, column 7, lines 58-65) and make the content of the annotations available to

tools such as checkers (see, for example, column 5, lines 2-10). Because the parsing of

annotations is compartmentalized and delegated to the annotation processors, Stata is able to

handle new types of annotations without changing the general parser (see, for example, column

7, lines 23-32).

One of ordinary skill in the art could, with predictable results, implement the teachings of

Necula such that the plug-in condition parses contents of a query string and makes the content

available to a checker as part of program's approximate execution state. As Stata suggests, such

an implementation would allow plug-in conditions to handle new types of query strings without

changing the checker. Thus, the claimed subject matter would have been obvious to one of

ordinary skill in the art at the time the invention was made.

Necula further discloses:

the checker that employs the specification to facilitate static checking of the object file,

the checker providing information if a fault condition is determined (see FIG. 4 – VCGen

module 32 – and associated text, e.g., column 13, lines 16-26: "The safety predicate 34 for a

function is obtained by evaluating it symbolically starting in a state that maps the global

variables … and function formal parameters to new variables …." FIGS. 3 and 5 – proof

checker 40 – and associated text, e.g., column 13, lines 25-35: "FIG. 5 is a diagram illustrating

an implementation of the proof checker module 40 of FIG. 3. The function of the proof checker

module 40 is to verify that the proof 38 supplied by the untrusted proof producer uses only

allowed axioms and inference rules and that it is a proof of the required safety predicate 34."

VCGen and the proof checker perform the checking of the executable statically based on the

provided specification.).

Necula does not explicitly disclose the checker passing a user injected custom state to the

plug-in condition to check the fault condition.

However, in an analogous art, Hallem teaches static analysis of code (see, for example,

page 69, Abstract). Hallem further teaches passing user-injected custom state values to

extensions or plug-in conditions that define state machines (see, for example, page 70, section

2.1, Metal Extensions and State Machines, first, second and third paragraphs) to check for bugs

and fault conditions (see, for example, page 69, Introduction, second paragraph). The teachings

of Hallem provide flexibility and ease of use in arbitrarily extending the checker (see, for

example, page 69, Introduction, third paragraph).

One of ordinary skill in the art could, with predictable results, implement the teachings of

Necula such that the checker passes a user injected custom state to the plug-in condition to check

the fault condition. As Hallem describes, such an implementation would provide flexibility and

ease of use in arbitrarily extending the checker of Necula. Thus, the claimed subject matter

would have been obvious to one of ordinary skill in the art at the time the invention was made.

With respect to claim 2 (original), Necula, Stata and Hallem disclose the system of claim

1, and Necula further discloses the plug-in condition comprising a precondition for the method

(see FIG. 4 – configuration data 50 – and associated text, e.g., column 7, lines 19-32: "The

configuration data 50 also describes, by precondition-postcondition pairs, all of the functions that

the untrusted code 42 are permitted to invoke.").

With respect to claim 3 (original), Necula, Stata and Hallem disclose the system of claim

2, and Necula further discloses the checker providing information associated with an object's

state after a call to the method, the information being based, at least in part, upon the plug-in

precondition (see FIG. 4 – VCGen module 32 – and associated text, e.g., column 13, lines 16-24:

"The safety predicate 34 for a function is obtained by evaluating it symbolically starting in a

state that maps the global variables … and function formal parameters to new variables ….").

With respect to claim 4 (original), Necula, Stata and Hallem disclose the system of claim

1, and Necula further discloses the plug-in condition comprising a postcondition for the method

(see FIG. 4 – configuration data 50 – and associated text, e.g., column 7, lines 19-32: "The

configuration data 50 also describes, by precondition-postcondition pairs, all of the functions that

the untrusted code 42 are permitted to invoke.").

With respect to claim 5 (original), Necula, Stata and Hallem disclose the system of claim

4, and Necula further discloses the checker providing information associated with an object's

state after a call to the method, the information being based, at least in part, upon the plug-in

postcondition (see FIG. 4 – VCGen module 32 – and associated text, e.g., column 13, lines 16-

24: "The safety predicate 34 for a function is obtained by evaluating it symbolically starting in a

state that maps the global variables … and function formal parameters to new variables ….").

With respect to claim 6 (original), Necula, Stata and Hallem disclose the system of claim

1, and Necula further discloses the object file being based, at least in part, upon a language that

compiles to Common Language Runtime (see, e.g., column 1, lines 40-55: "High level type-safe

programming languages, such as ML and Java … in practice programs often have some

components written in ML or Java and other components written in different languages (e.g. C or

assembly language)." According to the Meijer reference cited below, languages such as C

compile to the Common Language Runtime.)

With respect to claim 8 (original), Necula, Stata and Hallem disclose the system of claim

1, and Hallem further discloses the specification comprising information associated with a state-

machine protocol (see, for example, page 70, section 2.1, Metal Extensions and State Machines,

first paragraph).

With respect to claim 9 (original), Necula, Stata and Hallem disclose the system of claim 8, and Hallem further discloses a state of an object modeled with a custom state (see, for example, page 71, section 3.1, Metal States, second paragraph).

With respect to claim 10 (original), Necula, Stata and Hallem disclose the system of claim 9, and Hallem further discloses the state of the object further being modeled with a custom state component (see, for example, page 71, section 3.1, Metal States, fifth paragraph).

With respect to claim 11 (original), Necula, Stata and Hallem disclose the system of claim 10, and further disclose the specification comprising at least one of a plug-in precondition and a plug-in postcondition method, which is a method of the custom state that is invoked by the checker to perform interface-specific state checks and state transitions (see Necula, FIG. 4 – configuration data 50 – and associated text, e.g., column 7, lines 19-32: "The configuration data 50 also describes, by precondition-postcondition pairs, all of the functions that the untrusted code 42 are permitted to invoke." Hallem, pages 70-71, section 2.1, Metal Extensions and State Machines, sixth and seventh paragraphs: "Each state value defines a list of transitions. In the free checker, the start state defines a single transition rule and the v.freed state defines two.").

With respect to claim 12 (original), Necula, Stata and Hallem disclose the system of claim 1, and Necula further discloses wherein the specification is embedded with the object file (see FIG. 4 – annotated executable 30 – and associated text, e.g., column 5, line 66 to column 6, line 4. The annotations are embedded with the executable.).

With respect to claim 13 (original), Necula, Stata and Hallem disclose the system of claim 1, and Necula further discloses wherein the specification is stored in a specification repository (see FIG. 4 – configuration data 50 – and associated text, e.g., column 6, lines 55-59: "… a file of configuration data 50, which is provided as part of the safety policy by the code consumer." A file is stored in memory, e.g., in a file repository, and separate from the executable.).

With respect to claim 15 (currently amended), Necula discloses a method of facilitating static checking of executable code comprising:

receiving executable code (see FIG. 3 – annotated executable 30 – and associated text, e.g., column 4, lines 32-35);

receiving a specification associated with the executable code, the specification comprising information associated with at least one of a precondition or a postcondition for a method (see FIGS. 3 and 4 – VCGen module 32 – and associated text, e.g., column 4, lines 37-42: "The VCGen module 32 performs two tasks. First, it checks simple safety properties of the annotated executable 30. … Second, the VCGen module 32 watches for instructions whose execution might violate the safety policy." FIG. 4 – configuration data 50 – and associated text, e.g., column 7, lines 19-32: "The configuration data 50 also describes, by precondition-postcondition pairs, all of the functions that the untrusted code 42 are permitted to invoke." The examiner notes that the specification associated with the executable code is the annotation embedded in the executable and/or rules defined by the safety policy.).

Necula does not explicitly disclose that the at least one of a precondition or a postcondition parses contents of a query string and makes the content available to a checker as part of a program's approximate execution state.

However, in an analogous art, Stata teaches annotation processors that parse annotations (see, for example, column 7, lines 58-65) and make the content of the annotations available to tools such as checkers (see, for example, column 5, lines 2-10). Because the parsing of annotations is compartmentalized and delegated to the annotation processors, Stata is able to handle new types of annotations without changing the general parser (see, for example, column 7, lines 23-32).

One of ordinary skill in the art could, with predictable results, implement the teachings of Necula such that the at least one of a precondition or a postcondition parses contents of a query string and makes the content available to a checker as part of a program's approximate execution state. As Stata suggests, such an implementation would allow preconditions or postconditions to handle new types of query strings without changing the checker. Thus, the claimed subject matter would have been obvious to one of ordinary skill in the art at the time the invention was made.

Necula further discloses:

statically applying the specification to the executable code (see FIG. 4 – VCGen module 32 – and associated text, e.g., column 13, lines 16-24: "The safety predicate 34 for a function is obtained by evaluating it symbolically starting in a state that maps the global variables ... and function formal parameters to new variables ...." FIGS. 3 and 5 – proof checker 40 – and associated text, e.g., column 13, lines 25-35: "FIG. 5 is a diagram illustrating an implementation

of the proof checker module 40 of FIG. 3. The function of the proof checker module 40 is to

verify that the proof 38 supplied by the untrusted proof producer uses only allowed axioms and

inference rules and that it is a proof of the required safety predicate 34." VCGen and proof

checker perform the checking of the executable statically based on the provided specification).

Necula does not explicitly disclose passing a user injected custom state to the at least one

precondition or postcondition.

However, in an analogous art, Hallem teaches static analysis of code (see, for example,

page 69, Abstract). Hallem further teaches passing user-injected custom state values to

extensions or plug-in conditions that define state machines (see, for example, page 70, section

2.1, Metal Extensions and State Machines, first, second and third paragraphs) to check for bugs

and fault conditions (see, for example, page 69, Introduction, second paragraph). The teachings

of Hallem provide flexibility and ease of use in arbitrarily extending the checker (see, for

example, page 69, Introduction, third paragraph).

One of ordinary skill in the art could, with predictable results, implement the teachings of

Necula so as to pass a user injected custom state to the at least one precondition or postcondition.

As Hallem describes, such an implementation would provide flexibility and ease of use in

arbitrarily extending the checker of Necula. Thus, the claimed subject matter would have been

obvious to one of ordinary skill in the art at the time the invention was made.

Necula further discloses:

determining whether a fault condition exists based, at least in part, upon the statically

applied specification; and,

providing information associated with the fault condition, if a fault condition is

determined to exist.

(see FIG. 4 – safety predicate (SP) – and associated text, e.g., column 4, lines 40-48:

"The VCGen module 32 emits a predicate that expresses the conditions under which the

execution of the instruction is safe.")

With respect to claim 16 (original), Necula, Stata and Hallem disclose the method of

claim 15, and Necula further discloses a computer readable medium having stored thereon

computer executable instructions for carrying out the method of claim 15 (see FIG. 6 and column

16, lines 25-30: "Software verification modules 66, of the type disclosed herein in conjunction

with the present invention, are stored in the computer storage devices 64.").

With respect to claim 22 (currently amended), Necula discloses a computer readable

medium storing computer executable components of an executable code check system (see FIG.

6 and column 16, lines 25-30: "Software verification modules 66, of the type disclosed herein in

conjunction with the present invention, are stored in the computer storage devices 64.")

comprising:

an input component that receives an object file and a specification associated with the

object file (see FIGS. 3 and 4 – VCGen module 32 – and associated text, e.g., column 4, lines

37-42: "The VCGen module 32 performs two tasks. First, it checks simple safety properties of

the annotated executable 30. ... Second, the VCGen module 32 watches for instructions whose

execution might violate the safety policy." The examiner notes that the object file is

synonymous with the executable file, and the specification associated with the object file is the

annotation embedded in the executable and/or rules defined by the safety policy.),

the specification comprising information associated with a plug-in condition for a method

(see FIG. 4 – configuration data 50 – and associated text, e.g., column 7, lines 19-32: "The

configuration data 50 also describes, by precondition-postcondition pairs, all of the functions that

the untrusted code 42 are permitted to invoke.").

Necula does not explicitly disclose that the plug-in condition parses contents of a query

string and makes the content available to a checker component as part of a program's

approximate execution state.

However, in an analogous art, Stata teaches annotation processors that parse annotations

(see, for example, column 7, lines 58-65) and make the content of the annotations available to

tools such as checkers (see, for example, column 5, lines 2-10). Because the parsing of

annotations is compartmentalized and delegated to the annotation processors, Stata is able to

handle new types of annotations without changing the general parser (see, for example, column

7, lines 23-32).

One of ordinary skill in the art could, with predictable results, implement the teachings of

Necula such that the plug-in condition parses contents of a query string and makes the content

available to a checker component as part of program's approximate execution state. As Stata

suggests, such an implementation would allow plug-in conditions to handle new types of query

strings without changing the checker component. Thus, the claimed subject matter would have

been obvious to one of ordinary skill in the art at the time the invention was made.

Necula further discloses:

the checker component that employs the specification to facilitate static checking of the

object file, the checker component providing information if a fault condition is determined (see

FIG. 4 – VCGen module 32 – and associated text, e.g., column 13, lines 16-26: "The safety

predicate 34 for a function is obtained by evaluating it symbolically starting in a state that maps

the global variables … and function formal parameters to new variables …." FIGS. 3 and 5 –

proof checker 40 – and associated text, e.g., column 13, lines 25-35: "FIG. 5 is a diagram

illustrating an implementation of the proof checker module 40 of FIG. 3. The function of the

proof checker module 40 is to verify that the proof 38 supplied by the untrusted proof producer

uses only allowed axioms and inference rules and that it is a proof of the required safety

predicate 34." VCGen and the proof checker perform the checking of the executable statically

based on the provided specification.).

Necula does not explicitly disclose the checker component passing a user injected custom

state to the plug-in condition to check the fault condition.

However, in an analogous art, Hallem teaches static analysis of code (see, for example,

page 69, Abstract). Hallem further teaches passing user-injected custom state values to

extensions or plug-in conditions that define state machines (see, for example, page 70, section

2.1, Metal Extensions and State Machines, first, second and third paragraphs) to check for bugs

and fault conditions (see, for example, page 69, Introduction, second paragraph). The teachings

of Hallem provide flexibility and ease of use in arbitrarily extending the checker (see, for

example, page 69, Introduction, third paragraph).

One of ordinary skill in the art could, with predictable results, implement the teachings of

Necula such that the checker component passes a user injected custom state to the plug-in

condition to check the fault condition. As Hallem describes, such an implementation would

provide flexibility and ease of use in arbitrarily extending the checker of Necula. Thus, the

claimed subject matter would have been obvious to one of ordinary skill in the art at the time the

invention was made.

With respect to claim 23 (currently amended), Necula discloses a computer implemented

executable code check system (see FIG. 6) comprising:

means for receiving a specification associated with an object file, the specification

comprising information associated with a plug-in condition for a method (see FIGS. 3 and 4 –

VCGen module 32 – and associated text, e.g., column 4, lines 37-42: "The VCGen module 32

performs two tasks. First, it checks simple safety properties of the annotated executable 30. …

Second, the VCGen module 32 watches for instructions whose execution might violate the safety

policy." FIG. 4 – configuration data 50 – and associated text, e.g., column 7, lines 19-32: "The

configuration data 50 also describes, by precondition-postcondition pairs, all of the functions that

the untrusted code 42 are permitted to invoke." The examiner notes that the object file is

synonymous with the executable file, and the specification associated with the object file is the

annotation embedded in the executable and/or rules defined by the safety policy.).

Necula does not explicitly disclose that the plug-in condition parses contents of a query

string and makes the content available to a checker as part of a program's approximate execution

state.

However, in an analogous art, Stata teaches annotation processors that parse annotations

(see, for example, column 7, lines 58-65) and make the content of the annotations available to

tools such as checkers (see, for example, column 5, lines 2-10). Because the parsing of

annotations is compartmentalized and delegated to the annotation processors, Stata is able to

handle new types of annotations without changing the general parser (see, for example, column

7, lines 23-32).

One of ordinary skill in the art could, with predictable results, implement the teachings of

Necula such that the plug-in condition parses contents of a query string and makes the content

available to a checker as part of program's approximate execution state. As Stata suggests, such

an implementation would allow plug-in conditions to handle new types of query strings without

changing the checker. Thus, the claimed subject matter would have been obvious to one of

ordinary skill in the art at the time the invention was made.

Necula further discloses:

means for statically checking the object file based, at least in part, upon the specification

and determining if a fault condition exists (see FIG. 4 – VCGen module 32 – and associated text,

e.g., column 13, lines 16-26: "The safety predicate 34 for a function is obtained by evaluating it

symbolically starting in a state that maps the global variables … and function formal parameters

to new variables …." FIGS. 3 and 5 – proof checker 40 – and associated text, e.g., column 13,

lines 25-35: "FIG. 5 is a diagram illustrating an implementation of the proof checker module 40

of FIG. 3. The function of the proof checker module 40 is to verify that the proof 38 supplied by

the untrusted proof producer uses only allowed axioms and inference rules and that it is a proof

of the required safety predicate 34." VCGen and the proof checker perform the checking of the

executable statically based on the provided specification.).

Necula does not explicitly disclose means for passing a user injected custom state to the

plug-in condition.

However, in an analogous art, Hallem teaches static analysis of code (see, for example, page 69, Abstract). Hallem further teaches passing user-injected custom state values to extensions or plug-in conditions that define state machines (see, for example, page 70, section 2.1, Metal Extensions and State Machines, first, second and third paragraphs) to check for bugs and fault conditions (see, for example, page 69, Introduction, second paragraph). The teachings of Hallem provide flexibility and ease of use in arbitrarily extending the checker (see, for example, page 69, Introduction, third paragraph).

One of ordinary skill in the art could, with predictable results, implement the teachings of Necula so as to pass a user injected custom state to the plug-in condition. As Hallem describes, such an implementation would provide flexibility and ease of use in arbitrarily extending the checker of Necula. Thus, the claimed subject matter would have been obvious to one of ordinary skill in the art at the time the invention was made.

Necula further discloses:

means for providing information if a fault condition is determined to exist (see FIG. 4 – safety predicate (SP) – and associated text, e.g., column 4, lines 41-48: "… the VCGen module 32 emits a predicate that expresses the conditions under which the execution of the instruction is safe.").

With respect to claim 24 (currently amended), Necula discloses a method of performing static checking of executable code comprising:

receiving a request, the request including a parameter (see FIG. 4 – VCGen module 32 – and associated text, e.g., column 5, lines 60-65: "Both the precondition and postcondition are

parameters of the VCGen module 32 and are part of the safety policy." VCGen thus receives

precondition and postcondition parameters from configuration data file.).

Necula does not explicitly disclose receiving a plug-in condition that parses contents of a

query string and makes the content available to a checker as part of a program's approximate

execution state.

However, in an analogous art, Stata teaches annotation processors that parse annotations

(see, for example, column 7, lines 58-65) and make the content of the annotations available to

tools such as checkers (see, for example, column 5, lines 2-10). Because the parsing of

annotations is compartmentalized and delegated to the annotation processors, Stata is able to

handle new types of annotations without changing the general parser (see, for example, column

7, lines 23-32).

One of ordinary skill in the art could, with predictable results, implement the teachings of

Necula so as to receive a plug-in condition that parses contents of a query string and makes the

content available to a checker as part of program's approximate execution state. As Stata

suggests, such an implementation would allow plug-in conditions to handle new types of query

strings without changing the checker. Thus, the claimed subject matter would have been obvious

to one of ordinary skill in the art at the time the invention was made.

Necula further discloses:

setting a type of a result of a method call to a type of the parameter (The method or

function call is directly controlled by the parameter, i.e. preconditions and postconditions, and

thus they must be of the same type.); and

employing the parameter only during static checking of the method (see FIG. 4 – VCGen

module 32 – and associated text, e.g., column 13, lines 16-26: "The safety predicate 34 for a

function is obtained by evaluating it symbolically starting in a state that maps the global

variables … and function formal parameters to new variables …." FIGS. 3 and 5 – proof

checker 40 – and associated text, e.g., column 13, lines 25-35: "FIG. 5 is a diagram illustrating

an implementation of the proof checker module 40 of FIG. 3. The function of the proof checker

module 40 is to verify that the proof 38 supplied by the untrusted proof producer uses only

allowed axioms and inference rules and that it is a proof of the required safety predicate 34."

VCGen and the proof checker perform the checking of the executable statically based on the

provided specification.).

Necula does not explicitly disclose performing component-wise comparison of a user

injected custom state and a state defined by the parameter to determine a fault condition.

However, in an analogous art, Hallem teaches static analysis of code (see, for example,

page 69, Abstract). Hallem further teaches passing user-injected custom state values to

extensions or plug-in conditions that define state machines (see, for example, page 70, section

2.1, Metal Extensions and State Machines, first, second and third paragraphs) to check for bugs

and fault conditions (see, for example, page 69, Introduction, second paragraph). The teachings

of Hallem provide flexibility and ease of use in arbitrarily extending the checker (see, for

example, page 69, Introduction, third paragraph).

One of ordinary skill in the art could, with predictable results, implement the teachings of

Necula so as to perform component-wise comparison of a user injected custom state and a state

defined by the parameter to determine a fault condition. As Hallem describes, such an

implementation would provide flexibility and ease of use in arbitrarily extending the checker of

Necula. Thus, the claimed subject matter would have been obvious to one of ordinary skill in the

art at the time the invention was made.

4.   Claim 7 is rejected under 35 U.S.C. 103(a) as being unpatentable over Necula in view of

Stata and Hallem, as applied to claim 1 above, and further in view of Meijer et al., "Technical

Overview of the Common Language Runtime" (art of record, "Meijer").

        With respect to claim 7 (original), Necula, Stata and Hallem disclose the system of claim

1, but do not explicitly disclose the object file being based, at least in part, upon at least one of

C#, Visual Basic.net and Managed C++.

        However, Necula discloses that the executable, i.e. object file, is compiled from type-safe

and/or non-type-safe languages (see, e.g., column 1, lines 40-55: "High level type-safe

programming languages, such as ML and Java … in practice programs often have some

components written in ML or Java and other components written in different languages (e.g. C or

assembly language)."). Meijer discloses that the Common Language Runtime (CLR), which is

expressed in the Common Intermediate Language (CIL), can be compiled from a language such

as C, Pascal, C#, etc.(see, e.g., Introduction).

        Therefore, it would have been obvious to one of ordinary skill in the art at the time the

invention was made to use one of C#, Visual Basic.net and Managed C++ as a programming

language in Necula's invention because these statically typed object-oriented languages are well

supported by Microsoft's Common Language Infrastructure (CLI), as disclosed in Meijer, which

in turn provides strong support for CLR.

5.   Claim 14 is rejected under 35 U.S.C. 103(a) as being unpatentable over Necula in view of

Stata and Hallem, as applied to claim 1 above, and further in view of U.S. Patent No. 6,571,232

to Goldberg et al. (art of record, "Goldberg").

With respect to claim 14 (original), Necula, Stata and Hallem disclose the system of

claim 1, but do not explicitly disclose the system further comprising a specification extractor that

queries a database for its schema and stores information associated with the schema in a

specification repository.

However, Goldberg discloses a query object generator system comprising a specification

extractor that queries a database for its schema and stores information associated with the

schema in a specification repository (see FIG. 4 and associated text, e.g., column 6, lines 40-63:

"The query object generator tool 400 includes a mechanism (not shown) for obtaining the

database schema from database 404 ...." FIG. 6 and associated text, e.g. column 33, lines 5-13:

"The query object internal state object 602 allows the user to save a logical definition of a query

object in an intermediate file, such as file 612.").

Therefore, it would have been obvious to one of ordinary skill in the art at the time the

invention was made to allow the software verification system of Necula to query a database for

its schema and store information associated with it as disclosed in Goldberg so that the system

can further verify that command instructions in the program that are associated with database

querying are valid.

6.   Claims 17 and 18 are rejected under 35 U.S.C. 103(a) as being unpatentable over

Tichelaar et al., "A Metal-model for Language-Independent Refactoring" (art of record,

"Tichelaar") in view of Stata and in view of Hallem.

        With respect to claim 17 (currently amended), Tichelaar discloses a method of

developing a software component comprising:

        implementing a subclass of a custom state class (see FIG. 1: "Class" and

"InheritanceDefinition");

        implementing at least one of a plug-in precondition and a plug-in postcondition as a

method of the subclass (see Table 1: pre- and post-conditions in "Add Method").

        Tichelaar does not explicitly disclose that the at least one of a plug-in precondition or a

plug-in postcondition parses contents of a query string and makes the content available to a

checker as part of a program's approximate execution state.

        However, in an analogous art, Stata teaches annotation processors that parse annotations

(see, for example, column 7, lines 58-65) and make the content of the annotations available to

tools such as checkers (see, for example, column 5, lines 2-10). Because the parsing of

annotations is compartmentalized and delegated to the annotation processors, Stata is able to

handle new types of annotations without changing the general parser (see, for example, column

7, lines 23-32).

        One of ordinary skill in the art could, with predictable results, implement the teachings of

Tichelaar such that the at least one of a plug-in precondition or a plug-in postcondition parses

contents of a query string and makes the content available to a checker as part of a program's

approximate execution state. As Stata suggests, such an implementation would allow plug-in

preconditions or postconditions to handle new types of query strings without changing the

checker. Thus, the claimed subject matter would have been obvious to one of ordinary skill in

the art at the time the invention was made.

Tichelaar further discloses:

placing a custom attribute on an enclosing type declaration that references the custom

state sub class (see Table 1: "Add Attribute" and "Add Parameter" belonging to a subclass);

placing an attribute on a declaration that references the at least one of a plug-in

precondition or a plug-in postcondition (The pre- and post-conditions provide constraints for a

declaration of a class or subclass in Table 1. Thus, "Add Attribute" to the class or subclass with

the pre- and post-conditions as parameters would provide the declaration of that class or subclass

with a reference to the pre- and post-conditions.).

Tichelaar does not explicitly disclose determining a fault condition based upon the

information from the at least one of plug-in precondition or a plug-in postcondition.

However, in an analogous art, Hallem teaches static analysis of code (see, for example,

page 69, Abstract). Hallem further teaches extensions or plug-in conditions that define state

machines (see, for example, page 70, section 2.1, Metal Extensions and State Machines, first,

second and third paragraphs) that are employed to check for bugs and fault conditions (see, for

example, page 69, Introduction, second paragraph). The teachings of Hallem provide flexibility

and ease of use in arbitrarily extending the checker (see, for example, page 69, Introduction,

third paragraph).

One of ordinary skill in the art could, with predictable results, supplement the teachings

of Tichelaar so as to determine a fault condition based upon the information from the at least one

of plug-in precondition or a plug-in postcondition. Hallem describes a flexible and easily

extended checker that provides such determinations. Thus, the claimed subject matter would

have been obvious to one of ordinary skill in the art at the time the invention was made.

With respect to claim 18 (original), Tichelaar, Stata and Hallem disclose the method of

claim 17, but do not explicitly disclose a computer readable medium having stored thereon

computer executable instructions for carrying out the method of claim 17.

Nonetheless, it would have been obvious to one of ordinary skill in the art at the time the

invention was made that the method of claim 17 needs to be stored on a computer readable

medium in order for the functionality it is intended to perform to be realized by a computer.

7.   Claims 19 and 20 are rejected under 35 U.S.C. 103(a) as being unpatentable over Necula

in view of Stata.

With respect to claim 19 (currently amended), Necula discloses a method of performing

static checking of executable code comprising:

invoking a precondition plug-in that is arbitrary code written by a programmer, providing

the precondition plug-in with a program execution state at the call to a method (see, e.g., column

5, lines 50-60: "… the code consumer can declare a precondition, which is essentially a

description of the calling convention the consumer will use when invoking the untrusted code.");

receiving information from the precondition plug-in (see, e.g., column 5, lines 50-60);

Necula does not explicitly disclose that the precondition plug-in parses contents of a

query string and makes the content available to a checker as part of a program's approximate

execution state to enable the checker to find defects in the query.

However, in an analogous art, Stata teaches annotation processors that parse annotations (see, for example, column 7, lines 58-65) and make the content of the annotations available to tools such as checkers (see, for example, column 5, lines 2-10). Because the parsing of annotations is compartmentalized and delegated to the annotation processors, Stata is able to handle new types of annotations without changing the general parser (see, for example, column 7, lines 23-32).

One of ordinary skill in the art could, with predictable results, implement the teachings of Necula such that the precondition plug-in parses contents of a query string and makes the content available to a checker as part of a program's approximate execution state to enable the checker to find defects in the query. As Stata suggests, such an implementation would allow precondition plug-ins to handle new types of query strings without changing the checker. Thus, the claimed subject matter would have been obvious to one of ordinary skill in the art at the time the invention was made.

Necula further discloses:

determining whether a fault condition exists based, at least in part, upon the information from the pre-condition plug-in (see, e.g., column 7, lines 21-32: "The code consumer guarantees that the precondition holds when the untrusted code 42 is invoked. … The precondition for such a function is a predicate that the untrusted code 42 must establish before calling the function …."); and

providing information associated with the fault condition, if a fault condition is determined to exist (see FIG. 4 – safety predicate (SP) – and associated text, e.g., column 4, lines

40-48: "… the VCGen module 32 emits a predicate that expresses the conditions under which the execution of the instruction is safe.").

With respect to claim 20 (original), Necula and Stata disclose the method of claim 19, and Necula further discloses at least one of the following:

invoking a postcondition plug-in, providing the postcondition plug-in with the program execution state (see, e.g., column 5, lines 60-65: "… postconditions for the untrusted code. These are constraints on the final execution state of the untrusted code."); and

receiving information from the postcondition plug-in (see, e.g., column 7, lines 21-32: "The untrusted code 42 must ensure that the postcondition holds on return. … the postcondition is a predicate that the untrusted code 42 may assume to hold upon return from the function.")

Conclusion

8.   Any inquiry concerning this communication or earlier communications from the examiner should be directed to Michael J. Yigdall whose telephone number is 571-272-3707. The examiner can normally be reached on Monday to Friday from 8:00 AM to 4:30 PM.

If attempts to reach the examiner by telephone are unsuccessful, the examiner's supervisor, Tuan Q. Dam can be reached on 571-272-3695. The fax phone number for the organization where this application or proceeding is assigned is 571-273-8300.

Information regarding the status of an application may be obtained from the Patent

Application Information Retrieval (PAIR) system. Status information for published applications

may be obtained from either Private PAIR or Public PAIR. Status information for unpublished

applications is available through Private PAIR only. For more information about the PAIR

system, see http://pair-direct.uspto.gov. Should you have questions on access to the Private

PAIR system, contact the Electronic Business Center (EBC) at 866-217-9197 (toll-free). If you

would like assistance from a USPTO Customer Service Representative or access to the

automated information system, call 800-786-9199 (IN USA OR CANADA) or 571-272-1000.

                                                Michael J. Yigdall
                                                Examiner
                                                Art Unit 2192

/Michael J. Yigdall/
Examiner, Art Unit 2192